

Assignment-12.4

Task 1: Bubble Sort for Ranking Exam Scores

Scenario: You are working on a college result processing system where a small list of student scores needs to be sorted after every internal assessment.

Task Description

- Implement Bubble Sort in Python to sort a list of student scores.
- Use an AI tool to:
 - o Insert inline comments explaining key operations such as comparisons, swaps, and iteration passes
 - o Identify early-termination conditions when the list becomes sorted
 - o Provide a brief time complexity analysis

Expected Outcome

- A Bubble Sort implementation with:
 - o AI-generated comments explaining the logic
 - o Clear explanation of best, average, and worst-case complexity
 - o Sample input/output showing sorted scores give me a suitable prompt which includes everything.

The screenshot shows the GitHub Copilot interface with a code editor and a sidebar. The code editor contains a Python script named `task1.py` with the following content:

```
task1.py > ...
1 def bubble_sort(scores):
2     """
3         Implements Bubble Sort to sort student exam scores in ascending order.
4     Args:
5         scores: A list of integers representing student scores
6     Returns:
7         The sorted list in ascending order
8     """
9
10    n = len(scores)
11
12    # Outer loop: controls the number of passes through the list
13    for i in range(n):
14        # Flag to detect if any swaps occurred in this pass
15        # Used for early-termination optimization
16        swapped = False
17
18        # Inner loop: performs comparisons and swaps in current pass
19        # The range decreases by 1 each pass because the largest elements
20        # "bubble up" to their correct positions at the end of the list
21        for j in range(0, n - i - 1):
22            # Comparison: check if current score is greater than next score
23            if scores[j] > scores[j + 1]:
24                # Swap: exchange positions if elements are out of order
25                scores[j], scores[j + 1] = scores[j + 1], scores[j]
26                # Mark that a swap occurred in this pass
27                swapped = True
28
29        # Early-termination optimization: if no swaps occurred in this pass,
30        # the list is already sorted, so we can exit early
31        if not swapped:
32            break
33
34    return scores
35
36
37
38 # =====
39 # SAMPLE INPUT AND OUTPUT
```

The sidebar on the right displays the following information:

- TASK: IMPLEMENT BUBBLE SORT FOR RANKING EXAM SCORES**
 - The code should be clean and readable.
 - Comments must be technically clear and concise.
 - Do not overcomplicate the explanation.
 - Ensure the complexity analysis is accurate and properly formatted.
- Enable Claude Haiku 4.5 for all clients
- Enable access to the latest Claude Haiku 4.5 model from Anthropic. Learn more about how GitHub Copilot serves Claude Haiku 4.5.
- ✓ Selected "Enable"
- ✓ Reviewed and created task1.py
 - python task1.py ID ↗
- ✓ Completed implementation with detailed documentation and analysis
- Perfect! I've implemented a comprehensive Bubble Sort solution for ranking exam scores. Here's what's been completed:
- Implementation Highlights:**
 - ✓ Bubble sort function with inline comments explaining each comparison, swap, and pass through the list
 - ✓ Early-termination optimization using the `swapped` flag (exits when list is sorted)
 - ✓ Inner loop range optimization that reduces comparisons after each pass
- Describe what to build next

OUTPUT:

```
xe c:/Users/tloke/OneDrive/AI/task1.py
=====
BUBBLE SORT: STUDENT EXAM SCORES RANKING
=====

Original Score List: [85, 92, 78, 95, 88, 76, 90, 83]
Number of scores: 8
Sorted Score List: [76, 78, 83, 85, 88, 90, 92, 95]

=====
ALGORITHM EXPLANATION
=====

How Bubble Sort Works:
- Bubble Sort repeatedly steps through the list, comparing adjacent elements
- If an element is larger than the next element, they are swapped
- After each complete pass, the largest unsorted element "bubbles up"
  to its correct position at the end
- The process repeats until no swaps occur in a pass (list is sorted)
- Early-termination: If a pass produces no swaps, the algorithm stops
  as the list is already in order

Example with [85, 92, 78, 95]:
Pass 1: Compare pairs and swap if needed
  85 vs 92: No swap → [85, 92, 78, 95]
  92 vs 78: Swap → [85, 78, 92, 95]
  92 vs 95: No swap → [85, 78, 92, 95]
  Result: [85, 78, 92, 95] (95 is now in correct position)

Pass 2:
  85 vs 78: Swap → [78, 85, 92, 95]
  85 vs 92: No swap → [78, 85, 92, 95]
  Result: [78, 85, 92, 95] (no more swaps, list is sorted)

Pass 3: No swaps occur → Algorithm terminates (early exit)

=====
TIME COMPLEXITY ANALYSIS
=====

Best Case: O(n)
- Occurs when the list is already sorted
```

Task 2: Improving Sorting for Nearly Sorted Attendance Records

Scenario: You are maintaining an attendance system where student roll numbers are already almost sorted, with only a few late updates.

Task Description

- Start with a Bubble Sort implementation.
- Ask AI to:
 - o Review the problem and suggest a more suitable sorting algorithm

- o Generate an Insertion Sort implementation
- o Explain why Insertion Sort performs better on nearly sorted data
 - Compare execution behavior on nearly sorted input

Expected Outcome

- Two sorting implementations:

o Bubble Sort

o Insertion Sort

- AI-assisted explanation highlighting efficiency differences for partially sorted datasets

The screenshot shows a code editor with two panes. The left pane contains the Python code for `task2.py`, which implements both Bubble Sort and Insertion Sort for student roll numbers. The right pane is an AI-assisted explanation panel titled "TASK: IMPLEMENT BUBBLE SORT FOR RANKING EXAM SCORES". It includes a list of requirements, a "Key Findings" section comparing the two sorting algorithms, and a "Why Insertion Sort Excels for Nearly Sorted Data" section.

```

1  task2.py > ...
2  1 def bubble_sort_attendance(roll_numbers):
3  2     """
4  3     Implements Bubble Sort to sort student attendance roll numbers.
5  4
6  5     Args:
7  6         roll_numbers: List of integers representing student roll numbers
8  7
9  8     Returns:
10    9         Tuple of (sorted_list, comparison_count, swap_count)
11  10
12  11     n = len(roll_numbers)
13  12     comparisons = 0
14  13     swaps = 0
15
16  14     # Outer loop: controls number of passes through the list
17  15     for i in range(n):
18  16         # Flag for early-termination optimization
19  17         swapped = False
20
21  20         # Inner loop: compares adjacent elements in current pass
22  21         # Range decreases each pass as largest elements bubble to the end
23  22         for j in range(0, n - i - 1):
24  23             # Comparison: check if current roll number > next roll number
25  24             comparisons += 1
26  25             if roll_numbers[j] > roll_numbers[j + 1]:
27  26                 # Swap: exchange positions of elements out of order
28  27                 roll_numbers[j], roll_numbers[j + 1] = roll_numbers[j + 1], roll_
29  28                 swaps += 1
30  29                 swapped = True
31
32  30             # Early-termination optimization: if no swaps in this pass,
33  31             # the list is already sorted, so exit immediately
34  32             if not swapped:
35  33                 break
36
37  34     return roll_numbers, comparisons, swaps
38
39  35 def insertion_sort_attendance(roll_numbers):
36
37
38
39

```

TASK: IMPLEMENT BUBBLE SORT FOR RANKING EXAM SCORES

- Comments should be technically accurate and concise.
- Explanation should clearly highlight why Insertion Sort is more efficient for partially sorted datasets.
- Keep the explanation structured and easy to understand.

✓ Reviewed and created task1.py

python task1.py

Enable shell integration to improve command detection

Analyzed algorithm performance and provided comprehensive recommendations

Complete! I've created a comprehensive solution comparing Bubble Sort and Insertion Sort for nearly sorted attendance data.

Key Findings:

On the nearly sorted list with 20 roll numbers and 5 out-of-place entries:

- Bubble Sort:** 54 comparisons, 5 swaps
- Insertion Sort:** 24 comparisons, 5 shifts (55.6% more efficient)

Why Insertion Sort Excels for Nearly Sorted Data:

- Fewer Comparisons** – Only compares with sorted portion; stops when position found
- Early Detection** – Finds insertion point quickly in mostly-ordered data
- Minimal Work** – Shifts occur only for truly out-of-place elements

task2.py

Describe what to build next

task2.py

OUTPUT:

```
xe c:/Users/tloke/OneDrive/AI/task2.py
=====
SORTING NEARLY SORTED ATTENDANCE ROLL NUMBERS
=====

Nearly Sorted Roll Numbers (Input):
[1, 3, 2, 5, 6, 4, 7, 8, 9, 10, 11, 12, 13, 15, 14, 16, 17, 18, 20, 19]
Length: 20
```

```
Context: Most students present in order, but 5 out-of-place:
- 2 after 3 (should be 1,2,3...)
- 4 after 6 (should be 5,6,7...)
- 14 after 15 (should be 13,14,15...)
- 19 after 20 (should be 18,19,20...)
```

```
BUBBLE SORT EXECUTION
=====
```

```
Sorted Roll Numbers:
[1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12, 13, 14, 15, 16, 17, 18, 19, 20]
```

```
Bubble Sort Statistics:
- Comparisons: 54
- Swaps: 5
```

```
INSERTION SORT EXECUTION
=====
```

```
Sorted Roll Numbers:
[1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12, 13, 14, 15, 16, 17, 18, 19, 20]
```

```
Insertion Sort Statistics:
- Comparisons: 24
- Shifts: 5
```

```
WHY INSERTION SORT IS BETTER FOR NEARLY SORTED DATA
=====
```

```
Problem Characteristics:
```

| | | |
|---|-------------------------------|-----------------------|
| Comparisons (Nearly Sorted) | 54 | 24 |
| Shifts/Swaps (Nearly Sorted) | 5 | 5 |
| Best Case Complexity | $O(n)$ with early-termination | $O(n)$ |
| Average Case Complexity | $O(n^2)$ | $O(n^2)$ |
| Worst Case Complexity | $O(n^2)$ | $O(n^2)$ |
| Space Complexity | $O(1)$ | $O(1)$ |
| Stability | Yes | Yes |
| Adaptive (sorts faster if partially sorted) | No | Yes (Highly Adaptive) |
| Cache Efficient | Poor | Good |

```
PERFORMANCE SUMMARY FOR NEARLY SORTED DATA
=====
```

```
Test Data: 20 roll numbers, 5 out of place
```

```
Results:
```

```
  Bubble Sort:
    - Comparisons: 54
    - Swaps: 5
```

```
  Insertion Sort:
    - Comparisons: 24
    - Shifts: 5
```

```
Efficiency Gain:
```

```
  - Fewer comparisons with Insertion Sort: 55.6%
  - Insertion Sort is clearly superior for nearly sorted data
```

```
Recommendation:
```

Task 3: Searching Student Records in a Database

Scenario: You are developing a student information portal where users search for student records by roll number.

Task Description

- Implement:

- o Linear Search for unsorted student data

- o Binary Search for sorted student data

- Use AI to:

- o Add docstrings explaining parameters and return values

- o Explain when Binary Search is applicable

- o Highlight performance differences between the two searches

Expected Outcome

- Two working search implementations with docstrings

- AI-generated explanation of:

- o Time complexity

- o Use cases for Linear vs Binary Search

- A short student observation comparing results on sorted vs unsorted lists.

The image shows a code editor with a Python script named `task3.py`. The script contains code for linear and binary search algorithms, along with their respective docstrings and time complexity analysis. To the right of the code editor is a review interface for a task titled "IMPLEMENT BUBBLE SORT FOR RANKING EXAM SCORES". The review interface includes a checklist of requirements, a command history showing the execution of `python task3.py`, and a summary message stating that the script was created successfully. It also lists "Key Deliverables" and "Implementation Features" related to the search functions.

```
task3.py > ...
1 """
2 Student Record Search System
3
4 This module implements Linear Search and Binary Search algorithms
5 to demonstrate searching capabilities on student roll number databases.
6 It includes performance comparison and practical examples.
7 """
8
9
10 # =====
11 # LINEAR SEARCH IMPLEMENTATION
12 # =====
13
14 def linear_search(roll_numbers, target):
15     """
16     Search for a student roll number using Linear Search.
17
18     This function sequentially checks each element in the list until
19     the target is found or the end of the list is reached.
20     Works on both sorted and unsorted lists.
21
22     Parameters:
23     -----
24     roll_numbers : list
25         A list of integers representing student roll numbers (can be unsorted)
26     target : int
27         The roll number to search for
28
29     Returns:
30     -----
31     int
32         The index of the target roll number if found, -1 if not found
33
34     Time Complexity:
35     -----
36     Best case: O(1) - element found at first position
37     Average case: O(n) - element found at middle position
38     Worst case: O(n) - element at end or not present
39
```

TASK: IMPLEMENT BUBBLE SORT FOR RANKING EXAM SCORES

- Code must be clean, readable, and properly formatted.
- Docstrings must follow standard Python documentation style.
- Complexity analysis should be accurate.
- Explanation should be clear and suitable for an academic assignment.

✓ Reviewed and created task3.py

✓ o python task3.py ID R
Enable shell integration to improve command detection

✓ Reviewed script and confirmed successful execution results

Perfect! I've created a comprehensive Student Record Search System with both Linear and Binary Search implementations.

Key Deliverables:

- ✓ Linear Search: Works on unsorted data with O(n) worst-case
- ✓ Binary Search: Requires sorted data with O(log n) worst-case
- ✓ Both have proper Python docstrings (purpose, parameters, returns, complexity)
- ✓ Inline comments explaining every step of logic
- ✓ Helper function to verify if data is sorted

Test Results on 15-Student Dataset:

- Linear Search: 25 total comparisons, 6.2 average per search
- Binary Search: 15 total comparisons, 3.8 average per search

Describe what to build next

OUTPUT:

```

xe c:/Users/tloke/OneDrive/AI/task3.py
=====
STUDENT RECORD SEARCH SYSTEM
Linear Search vs Binary Search Comparison
=====

=====
INPUT DATA
=====

Unsorted Roll Numbers (15 students):
[105, 201, 42, 156, 89, 233, 67, 178, 145, 56, 219, 33, 198, 112, 76]
Status: Not Sorted

Sorted Roll Numbers (15 students):
[33, 42, 56, 67, 76, 89, 105, 112, 145, 156, 178, 198, 201, 219, 233]
Status: Sorted

Search Targets: [105, 150, 233, 42]

=====
LINEAR SEARCH RESULTS (Searching in Unsorted Data)
=====

Linear Search checks elements sequentially until found or end reached.
Works on unsorted data.

Target: 105 → Found at index 0 | Comparisons: 1 | Roll at index: 105
Target: 150 → NOT FOUND | Comparisons: 15
Target: 233 → Found at index 5 | Comparisons: 6 | Roll at index: 233
Target: 42 → Found at index 2 | Comparisons: 3 | Roll at index: 42

Total comparisons (Linear Search): 25
Average comparisons per search: 6.2

=====
BINARY SEARCH RESULTS (Searching in Sorted Data)
=====

Binary Search divides search space in half with each comparison.
REQUIRES sorted data in ascending order.

```

```

=====
TIME AND SPACE COMPLEXITY COMPARISON
=====



| Metric               | Linear Search                | Binary Search                      |
|----------------------|------------------------------|------------------------------------|
| Best Case            | $O(1)$ - Found at start      | $O(1)$ - Found in middle           |
| Average Case         | $O(n)$ - Found at middle     | $O(\log n)$ - Halving search space |
| Worst Case           | $O(n)$ - Not found or at end | $O(\log n)$ - Not found            |
| Space Complexity     | $O(1)$ - No extra space      | $O(1)$ - No extra space            |
| Requires Sorted Data | No - Works on unsorted       | Yes - MUST be sorted               |
| Data Structure       | Array or List                | Must support indexing              |



=====
PRACTICAL PERFORMANCE ANALYSIS
=====

Test Results Summary:
-----
Linear Search:


- Total comparisons for 4 searches: 25
- Average per search: 6.2
- Can search unsorted data directly
- Consistent performance regardless of data order


Binary Search:


- Total comparisons for 4 searches: 15
- Average per search: 3.8
- Requires sorted data (one-time sorting cost)
- Significantly fewer comparisons on large datasets


-----
For this 15-element dataset:
Efficiency Gain: 40.0% fewer comparisons with Binary Search

Scaling Example (1000 elements):
Linear Search: Up to 1000 comparisons in worst case
Binary Search: Up to ~10 comparisons in worst case
Advantage: Binary Search ~100x faster!

```

Task 4: Choosing Between Quick Sort and Merge Sort for Data Processing

Scenario: You are part of a data analytics team that needs to sort large datasets received from different sources (random order, already sorted, and reverse sorted).

Task Description

- Provide AI with partially written recursive functions for:

- o Quick Sort

- o Merge Sort

- Ask AI to:

- o Complete the recursive logic

- o Add meaningful docstrings

- o Explain how recursion works in each algorithm

- Test both algorithms on:

- o Random data

- o Sorted data

- o Reverse-sorted data

Expected Outcome

- Fully functional Quick Sort and Merge Sort implementations

- AI-generated comparison covering:

- o Best, average, and worst-case complexities Practical scenarios where one algorithm is preferred over the other

The screenshot shows a code editor with two panes. The left pane contains the Python code for `task4.py`. The right pane is a task summary titled "TASK: IMPLEMENT BUBBLE SORT FOR RANKING EXAM SCORES". It includes a checklist, a summary message, and implementation highlights for Quick Sort and Merge Sort.

```

task4.py > ...
1 """
2 Quick Sort vs Merge Sort: Comparison for Large Datasets
3
4 This module implements and compares Quick Sort and Merge Sort algorithms
5 on different data patterns: random, sorted, and reverse-sorted.
6 Includes time complexity analysis and practical recommendations.
7 """
8
9 import random
10 import time
11
12 # =====
13 # QUICK SORT IMPLEMENTATION
14 # =====
15
16 def quick_sort(arr, low=0, high=None):
17     """
18         Sorts an array using Quick Sort algorithm (Divide and Conquer).
19
20         Quick Sort selects a pivot element and partitions the array into
21         elements smaller than pivot (left) and greater than pivot (right).
22         Then recursively sorts both partitions.
23
24     Parameters:
25     -----
26     arr : list
27         The array/list to be sorted (modified in-place)
28     low : int
29         Starting index for current partition (default: 0)
30     high : int
31         Ending index for current partition (default: len(arr) - 1)
32
33     Returns:
34     -----
35     None (sorts in-place)
36
37 Start
38     Time Complexity:

```

TASK: IMPLEMENT BUBBLE SORT FOR RANKING EXAM SCORES

- Provide comparison including:
 - Best, average, worst-case time complexity
 - Space complexity
 - Practical scenarios where Quick Sort or Merge Sort is preferred.
 - Keep code clean and explanation concise but accurate.

✓ Reviewed and created task4.py

✓ python task4.py

✓ Enable shell integration to improve command detection

✓ Reviewed content from specified lines

Perfect! I've created a comprehensive Quick Sort and Merge Sort comparison implementation. Here's a summary of what was accomplished:

Implementation Highlights:

✓ **Quick Sort Implementation:**

- Complete recursive partitioning logic using Lomuto scheme
- Proper docstrings explaining purpose, parameters, returns, and complexity
- Inline comments explaining recursion and pivot placement
- Time: $O(n \log n)$ average, $O(n^2)$ worst case
- Space: $O(\log n)$ - sorts in-place

✓ **Merge Sort Implementation:**

- Complete recursive divide-and-conquer logic
- Proper merge function combining sorted subarrays
- Docstrings with detailed complexity analysis

OUTPUT:

```

=====
QUICK SORT vs MERGE SORT: COMPARISON ON LARGE DATASETS
=====

=====
EXECUTION RESULTS
=====

TEST SIZE: 100 elements
=====

RANDOM DATA
-----
Sample Input: [485, 403, 185, 320, 721, 342, 628, 529, 982, 922, ..., 336, 120, 158, 677, 725]
Quick Sort → Time: 0.115ms | Sorted: √
Merge Sort → Time: 0.194ms | Sorted: √
Faster: Quick Sort (40.5% difference)

SORTED DATA
-----
Sample Input: [1, 2, 3, 4, 5, 6, 7, 8, 9, 10, ..., 96, 97, 98, 99, 100]
Quick Sort → Time: 0.338ms | Sorted: √
Merge Sort → Time: 0.132ms | Sorted: √
Faster: Merge Sort (61.0% difference)

REVERSE-SORTED DATA
-----
Sample Input: [100, 99, 98, 97, 96, 95, 94, 93, 92, 91, ..., 5, 4, 3, 2, 1]
Quick Sort → Time: 0.229ms | Sorted: √
Merge Sort → Time: 0.090ms | Sorted: √
Faster: Merge Sort (60.7% difference)

```

| TIME AND SPACE COMPLEXITY COMPARISON | | | |
|--------------------------------------|-----------------|----------------------|--------------------------------------|
| Scenario | Quick Sort | Merge Sort | Notes |
| Best Case | $O(n \log n)$ | $O(n \log n)$ | QS: pivot divides evenly |
| Average Case | $O(n \log n)$ | $O(n \log n)$ | Both are consistent |
| Worst Case | $O(n^2)$ | $O(n \log n)$ | QS: pivot at end (reversed) |
| Space Complexity | $O(\log n)$ | $O(n)$ | QS: recursive stack; MS: temp arrays |
| In-Place Sort | Yes | No | MS requires extra memory |
| Stable Sort | No | Yes | MS preserves relative order |
| Cache Efficiency | Better | Worse | QS: sequential access pattern |
| Best For | Random/avg data | Worst-case guarantee | See scenarios below |

PRACTICAL SCENARIOS: WHEN TO USE WHICH ALGORITHM

USE QUICK SORT WHEN:

- ✓ Data is randomly distributed (most common case)
- ✓ Memory usage is critical (sorts in-place)
- ✓ Average-case $O(n \log n)$ performance is acceptable
- ✓ You have fast CPU but limited RAM
- ✓ Real-world: Standard library sorts (`std::sort` in C++), Java `Arrays.sort()`
- ✓ Real-world: Quickselect algorithm for finding k-th element

Examples:

- Sorting student grades (random distribution)

Task 5: Optimizing a Duplicate Detection Algorithm

Scenario: You are building a data validation module that must detect duplicate user IDs in a large dataset before importing it into a system.

Task Description

- Write a naive duplicate detection algorithm using nested loops.
- Use AI to:
 - o Analyze the time complexity

- o Suggest an optimized approach using sets or dictionaries
 - o Rewrite the algorithm with improved efficiency
 - Compare execution behavior conceptually for large input sizes
- Expected Outcome**
- Two versions of the algorithm:
 - o Brute-force ($O(n^2)$)
 - o Optimized ($O(n)$)
 - AI-assisted explanation showing how and why performance improved

The screenshot shows a code editor with a Python file named `task5.py`. The file contains code for detecting duplicates in user ID datasets, comparing a naive $O(n^2)$ approach with an optimized $O(n)$ set-based approach. The AI-assisted explanation sidebar provides details on the task, highlights for both approaches, and a summary of the optimized version's benefits.

```

task5.py > ...
1 """
2 Duplicate Detection Optimization: Naive vs Optimized Approaches
3
4 This module compares a naive  $O(n^2)$  nested-loop approach with an optimized
5  $O(n)$  set-based approach for detecting duplicates in large user ID datasets.
6 """
7
8 import time
9
10 # =====
11 # NAIVE APPROACH:  $O(n^2)$  TIME
12 # =====
13
14 def detect_duplicates_naive(user_ids):
15     """
16     Naive approach to detect duplicates using nested loops.
17
18     WARNING: This function is INEFFICIENT for large datasets.
19     Time Complexity:  $O(n^2)$  - very slow!
20     Space Complexity:  $O(1)$  - only uses constant extra space
21
22     This is included for educational purposes to show the performance
23     difference between naive and optimized approaches.
24
25     Parameters:
26     -----
27     user_ids : list
28         List of integers representing user IDs
29
30     Returns:
31     -----
32     list
33         A list of duplicate user IDs found, or empty list if no duplicates
34
35     How it works:
36     - For each user ID, compare it with ALL other user IDs
37     - If a match is found, add to duplicates list
38

```

TASK: IMPLEMENT BUBBLE SORT FOR RANKING EXAM SCORES

- o Sample input/output
- o Time complexity comparison ($O(n^2)$ vs $O(n)$)
- o Clear explanation of why the optimized version performs better.

Keep the code clean, efficient, and explanations concise but accurate.

✓ Reviewed and created Task5.py

✓ python Task5.py

✓ Enable shell integration to improve command detection

✓ Reviewed content from specified lines

Excellent! I've created a comprehensive Duplicate Detection Optimization solution. Here are the key results:

Implementation Highlights:

✓ **Naive Approach ($O(n^2)$):**

- Uses nested loops to compare every pair of user IDs
- Clear inline comments explaining the algorithm
- Time: $O(n^2)$ - very inefficient for large datasets
- Space: $O(1)$ - minimal extra memory

✓ **Optimized Approach ($O(n)$):**

- Uses a set for $O(1)$ lookups
- Proper Python docstrings with complexity analysis
- Single pass through the data
- Space: $O(n)$ - reasonable trade-off

✓ **Alternative Approach:**

Describe what to build next

OUTPUT:

```
xe c:/Users/tloke/OneDrive/AI/task5.py
=====
=====
DUPLICATE DETECTION OPTIMIZATION: NAIVE vs OPTIMIZED
=====

=====
TEST CASE 1: SMALL DATASET (20 user IDs)
=====

=====
Input: [1001, 1005, 1003, 1007, 1005, 1002, 1008, 1005, 1001, 1009, 1004, 1003, 1010, 1012, 1007
, 1012, 1015, 1020, 1001, 1025]
List length: 20 user IDs
Expected duplicates: 1001 (3 times), 1005 (3 times), 1003 (2 times), 1007 (2 times), 1012 (2 times)

-----
NAIVE APPROACH (Nested Loops - O(n²)):
Duplicates found: [1001, 1005, 1003, 1007, 1012]
Execution time: 0.0341 ms
Comparisons made: ~190 (theoretical)

-----
OPTIMIZED APPROACH (Set - O(n)):
Duplicates found: [1001, 1003, 1005, 1007, 1012]
Execution time: 0.0184 ms
Comparisons made: ~20 (single pass)

-----
ALTERNATIVE: DICT APPROACH (with counts):
Duplicates with counts: {1001: 3, 1005: 3, 1003: 2, 1007: 2, 1012: 2}
Execution time: 0.0260 ms

=====
TEST CASE 2: MEDIUM DATASET (1,000 user IDs)
```

TIME COMPLEXITY COMPARISON

| Metric | Naive (Nested Loops) | Optimized (Set) |
|-------------------------|--------------------------|--------------------|
| Time Complexity | $O(n^2)$ | $O(n)$ |
| Space Complexity | $O(1)$ | $O(n)$ |
| Operations for n=100 | ~4,950 comparisons | ~100 lookups |
| Operations for n=1,000 | ~499,500 comparisons | ~1,000 lookups |
| Operations for n=10,000 | ~499,500,000 comparisons | ~10,000 lookups |
| Suitable for small data | Yes (< 100 items) | Always optimal |
| Suitable for large data | NO - becomes unusable | Yes - fast always |
| Memory overhead | Minimal | Extra $O(n)$ space |

WHY THE OPTIMIZED APPROACH PERFORMS BETTER

1. FUNDAMENTAL DIFFERENCE IN ALGORITHM:

NAIVE (Nested Loops):

- For each element (n iterations)
- Compare with all other elements (n comparisons per element)
- Total: $n \times n = n^2 = O(n^2)$ operations
- Example: 1,000 items = 1,000,000 comparisons!

OPTIMIZED (Set-Based):

- Iterate through each element once (n iterations)
- Each lookup/insertion in set is $O(1)$ on average
- Total: $n \times 1 = O(n)$ operations
- Example: 1,000 items = 1,000 operations!

2. MATHEMATICAL SCALING:

| Data Size | Naive Ops | Optimized Ops | Speedup |
|-----------|-----------|---------------|---------------|
| 100 | 4,950 | 100 | 49.5x faster |
| 1,000 | 499,500 | 1,000 | 499.5x faster |

